White Box Testing

Girish Janardhanudu, Cigital, Inc. [vita[3]]

2005-09-26; Updated 2009-07-27 by Ken van Wyk [vita[4]]

L2/L3 / L, M[5]

White box testing is a security testing method that can be used to validate whether code implementation follows intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities.

Overview

This paper introduces white box testing for security, how to perform white box testing, and tools and techniques relevant to white box testing. It brings together concepts from two separate domains: traditional white box testing techniques and security testing. It assumes the reader to be familiar with general concepts of software security. Refer to other content areas on this portal to learn different aspects of software security.

This paper will help security developers and testers understand white box testing for security and how to effectively use the approach, tools, and techniques applicable to white box testing.

The paper is organized into separate sections dealing with what white box testing is, how to perform white box testing, what results to expect, the business case to justify white box testing, skills and training required to perform white box testing, and a brief case study.

What is White Box Testing?

The purpose of any security testing method is to ensure the robustness of a system in the face of malicious attacks or regular software failures. White box testing is performed based on the knowledge of *how* the system is implemented. White box testing includes analyzing data flow, control flow, information flow,

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/257-BSI.html (Janardhanudu, Girish)
6. #dsy259-BSI_over
7. #dsy259-BSI_whatis
8. #dsy259-BSI_BandG
9. #dsy259-BSI_howto
10. #dsy259-BSI_results
11. #dsy259-BSI_buscase
12. #dsy259-BSI_skills
13. #dsy259-BSI_study
14. #dsy259-BSI_conclusion
15. #dsy259-BSI_links
16. #dsy259-BSI_glossary

coding practices, and exception and error handling within the system, to test the intended and unintended software behavior. White box testing can be performed to validate whether code implementation follows intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities.

White box testing requires access to the source code. Though white box testing can be performed any time in the life cycle after the code is developed, it is a good practice to perform white box testing during the unit testing phase.

White box testing requires knowing what makes software secure or insecure, how to think like an attacker, and how to use different testing tools and techniques. The first step in white box testing is to comprehend and analyze available design documentation, source code, and other relevant development artifacts, so knowing what makes software secure is a fundamental requirement. Second, to create tests that exploit software, a tester must think like an attacker. Third, to perform testing effectively, testers need to know the different tools and techniques available for white box testing. The three requirements do not work in isolation, but together.

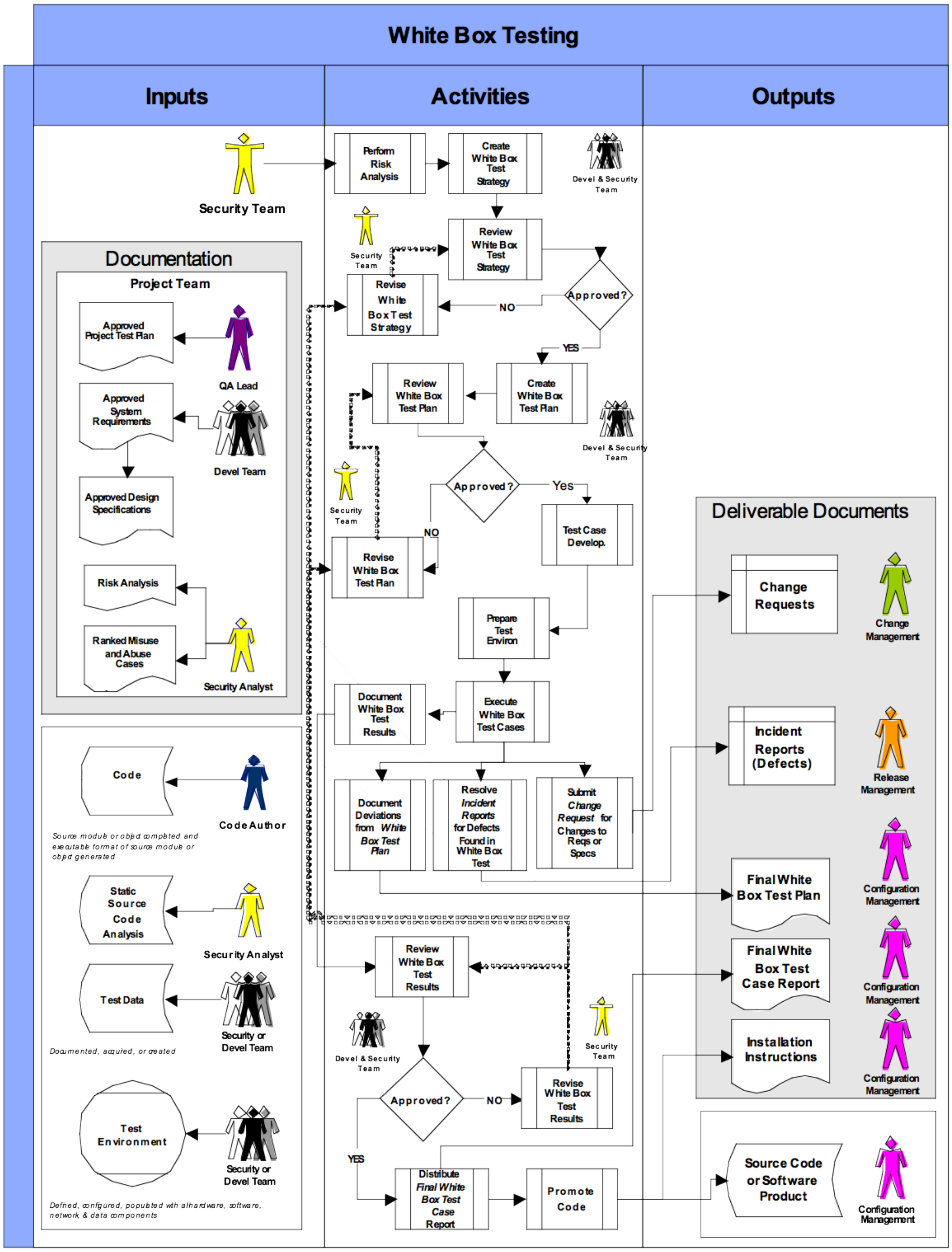
Black Box and Gray Box Testing

Black box testing is based on the software’s specifications or requirements, without reference to its internal workings. Gray box testing combines white box techniques with black box input testing [Hoglund 04]. This method of testing explores paths that are directly accessible from user inputs or external interfaces to the software. In a typical case, white box analysis is used to find vulnerable areas, and black box testing is then used to develop working attacks against these areas. The use of gray box techniques combines both white box and black box testing methods in a powerful way.

How to Perform White Box Testing

Process

Figure 1. White box testing process


White Box Testing
Inputs
Activities
Outputs
Security Team
Documentation
Project Team
Approved Project Test Plan
QA Lead
Approved System Requirements
Devel Team
Approved Design Specifications
Risk Analysis
Ranked Misuse and Abuse Cases
Security Analyst
Code
Code Author
Source module or object completed and executable format of source module or object generated
Static Source Code Analysis
Security Analyst
Test Data
Security or Devel Team
Documented, acquired, or created
Test Environment
Security or Devel Team
Defined, configured, populated with all hardware, software, network & data components
Perform Risk Analysis
Create White Box Test Strategy
Devel & Security Team
Security Team
Review White Box Test Strategy
Revise White Box Test Strategy
NO
Approved?
YES
Review White Box Test Plan
Create White Box Test Plan
Devel & Security Team
Security Team
Approved?
Yes
NO
Revise White Box Test Plan
Test Case Develop.
Prepare Test Environ
Document White Box Test Results
Execute White Box Test Cases
Document Deviations from White Box Test Plan
Resolve Incident Reports for Defects Found in White Box Test
Submit Change Request for Changes to Reqs or Specs
Review White Box Test Results
Devel & Security Team
Security Team
Approved?
NO
Revise White Box Test Results
YES
Distribute Final White Box Test Case Report
Promote Code
Deliverable Documents
Change Requests
Change Management
Incident Reports (Defects)
Release Management
Final White Box Test Plan
Configuration Management
Final White Box Test Case Report
Configuration Management
Installation Instructions
Configuration Management
Source Code or Software Product
Configuration Management

The figure provides a graphic depiction of the security testing process. This same process applies at all levels of testing, from unit testing to systems testing. The use of this document does not require subscribing to a specific testing process or methodology. Readers are urged to fit the activities described here into the process followed within their organization.

The general outline of the white box testing process is as follows:

1. Perform risk analysis to guide the whole testing process. In Microsoft's SDL process, this is referred to as threat modeling [Howard 06].
2. Develop a test strategy that defines what testing activities are needed to accomplish testing goals.
3. Develop a detailed test plan that organizes the subsequent testing process.
4. Prepare the test environment for test execution.
5. Execute test cases and communicate results.
6. Prepare a report.

In addition to the general activities described above, the process diagram introduces review cycles, reporting mechanisms, deliverables, and responsibilities.

The following sections discuss inputs, activities, and deliverable outputs in detail.

Inputs

Some of the artifacts relevant to white box testing include source code, a risk analysis report, security specification/requirements documentation, design documentation, and quality assurance related documentation.

- Source code is the most important artifact needed to perform white box testing. Without access to the code, white box testing cannot be performed, since it is based on testing software knowing *how* the system is implemented. (In theory, white box testing could also be performed on disassembled or decompiled code, but the process would likely be excessively labor intensive and error prone.)
- Architectural and design risk analysis should be the guiding force behind all white box testing related activities, including test planning, test case creation, test data selection, test technique selection, and test exit criteria selection. If a risk analysis was not completed for the system, this should be the first activity performed as part of white box testing. The following section discusses risk analysis.
- Design documentation is essential to improve program understanding and to develop effective test cases that validate design decisions and assumptions. If design documentation is unavailable or otherwise insufficient, the test team will at a minimum require direct access to the design team for extensive question and answer sessions. Either way, the test team will need to build a detailed understanding of how the software should behave.
- Security specifications or requirements are a must, to understand and validate the security functionality of the software under test. Similarly, if the security requirements and specifications are lacking, the test team will need to obtain this information from various stakeholders, including the design team as well as the software's business owner.
- Security testers should have access to quality assurance documentation to understand the quality of the software with respect to its intended functionality. Quality assurance documentation should include a test strategy, test plans, and defect reports. Load and performance tests are important in understanding the constraints placed on the system and the behavior of the system under stress.
- Any artifact relevant to program understanding should be available to white box testers.

Risk Analysis

Security is always relative to the information and services being protected, the skills and resources of adversaries, and the costs of potential assurance remedies; security is an exercise in risk management. The object of risk analysis is to determine specific vulnerabilities and threats that exist for the software and assess their impact. White box testing should use a risk-based approach, grounded in both the system's implementation and the attacker's mindset and capabilities.

White box testing should be based on architecture and design-level risk analysis. This content area will discuss how to use the results of risk analysis for white box testing, while the Architectural Risk Analysis[51] content area discusses risk analysis in detail.

Risk analysis should be the guiding force behind all white box testing related activities. The following paragraphs briefly introduce how the risk analysis results are used in white box testing. The subsequent sections discuss the activities in detail.

The risk analysis report, in combination with a functional decomposition of the application into major components, processes, data stores, and data communication flows, mapped against the environments across which the software will be deployed, allows for a desktop review of threats and potential vulnerabilities. The risk analysis report should help identify

- the threats present in each tier (or components)
- the kind of vulnerabilities that might exist in each component
- the business impact (consequence and cost of failure of software) of risks
- the probability (likelihood) of the risks being realized
- existing and recommended countermeasures to mitigate identified risks

Use the above information from the risk analysis report to

- develop a test strategy: Exhaustive testing is seldom cost-effective and often not possible in finite time. Planned testing is therefore selective, and this selection should be based on risks to the system. The priority (or ranking) of risks from the risk analysis should be the guiding rule for the focus of testing, simply because highly vulnerable areas should be tested thoroughly. The test strategy captures all the decisions, priorities, activities, and focus of testing based on the consequence of failure of software. The following section discusses test strategy in detail. For detailed research on risk-based test planning, refer to [Redmill 04].
- develop test cases: While a test strategy targets the overall test activities based on risks to the system, a test case can target specific concerns or risks based on the threats, vulnerabilities, and assumptions uncovered during the analysis. For example, tests can be developed to validate controls (or safeguards) put in place to mitigate a certain risk.
- determine test coverage: The higher the consequence of failure of certain areas (or components), the higher the test coverage should be in those areas. Risk-based testing allows for justifying the rigor of testing in a particular area. For example, a specific component or functionality may have high exposure to untrusted inputs, hence warranting extra testing attention.

Test Strategy

The first step in planning white box testing is to develop a test strategy based on risk analysis. The purpose of a test strategy is to clarify the major activities involved, key decisions made, and challenges faced in the testing effort. This includes identifying testing scope, testing techniques, coverage metrics, test environment, and test staff skill requirements. The test strategy must account for the fact that time and budget constraints prohibit testing every component of a software system and should balance test effectiveness with test efficiency based on risks to the system. The level of effectiveness necessary depends on the use of software and its consequence of failure. The higher the cost of failure for software, the more sophisticated and rigorous a testing approach must be to ensure effectiveness. Risk analysis provides the right context and information to derive a test strategy.

Test strategy is essentially a management activity. A test manager (or similar role) is responsible for developing and managing a test strategy. The members of the development and testing team help the test manager develop the test strategy.

51. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/architecture.html (Architectural Risk Analysis)

Test Plan

The test plan should manifest the test strategy. The main purpose of having a test plan is to organize the subsequent testing process. It includes test areas covered, test technique implementation, test case and data selection, test results validation, test cycles, and entry and exit criteria based on coverage metrics. In general, the test plan should incorporate both a high-level outline of which areas are to be tested and what methodologies are to be used and a general description of test cases, including prerequisites, setup, execution, and a description of what to look for in the test results. The high-level outline is useful for administration, planning, and reporting, while the more detailed descriptions are meant to make the test process go smoothly.

While not all testers like using test plans, they provide a number of benefits:

- Test plans provide a written record of what is to be done.
- Test plans allow project stakeholders to sign off on the intended testing effort. This helps ensure that the stakeholders agree with and will continue to support the test effort.
- Test plans provide a way to measure progress. This allows testers to determine whether they are on schedule, and also provides a concise way to report progress to the stakeholders.
- Due to time and budget constraints, it is often impossible to test all components of a software system. A test plan allows the analyst to succinctly record what the testing priorities are.
- Test plans provide excellent documentation for testing subsequent releases—they can be used to develop regression test suites and/or provide guidance to develop new tests.

A test manager (or similar role) is responsible for developing and managing a test plan. The development managers are also part of test plan development, since the schedules in the test plan are closely tied to that of the development schedules.

Test Case Development

The last activity within the test planning stage is test case development. Test case description includes preconditions, generic or specific test inputs, expected results, and steps to perform to execute the test. There are many definitions and formats for test case description. In general, the intent of the test case is to capture what the particular test is designed to accomplish. Risk analysis, test strategy, and the test plan should guide test case development. Testing techniques and classes of tests applicable to white box testing are discussed later in Sections 4.2 and 4.3 respectively.

The members of the testing team are responsible for developing and documenting test cases.

Test Automation

Test automation provides automated support for the process of managing and executing tests, especially for repeating past tests. All the tests developed for the system should be collected into a test suite. Whenever the system changes, the suite of tests that correspond to the changes or those that represent a set of regression tests can be run again to see if the software behaves as expected. Test drivers or suite drivers support executing test suites. Test drivers basically help in setup, execution, assertion, and teardown for each of the tests. In addition to driving test execution, test automation requires some automated mechanisms to generate test inputs and validate test results. The nature of test data generation and test results validation largely depends on the software under test and on the testing intentions of particular tests.

In addition to test automation development, stubs or scaffolding development is also required. Test scaffolding provides some of the infrastructure needed in order to test efficiently. White box testing mostly requires some software development to support executing particular tests. This software establishes an environment around the test, including states and values for data structures, runtime error injection, and acts as stubs for some external components. Much of what scaffolding is for depends on the software under test. However, as a best practice, it is preferable to separate the test data inputs from the code that delivers them, typically by putting inputs in one or more separate data files. This simplifies test maintenance and allows for reuse of test code. The members of the testing team are responsible for test automation and supporting software development. Typically, a member of the test team is dedicated to the development effort.

Test Environment

Testing requires the existence of a test environment. Establishing and managing a proper test environment is critical to the efficiency and effectiveness of testing. For simple application programs, the test environment generally consists of a single computer, but for enterprise-level software systems, the test environment is much more complex, and the software is usually closely coupled to the environment.

For security testing, it is often necessary for the tester to have more control over the environment than in many other testing activities. This is because the tester must be able to examine and manipulate software/ environment interactions at a greater level of detail, in search of weaknesses that could be exploited by an attacker. The tester must also be able to control these interactions. The test environment should be isolated, especially if, for example, a test technique produces potentially destructive results during test that might invalidate the results of any concurrent test or other activity. Testing malware (malicious software) can also be dangerous without strict isolation.

The test manager is responsible for coordinating test environment preparation. Depending on the type of environment required, the members of the development, testing, and build management teams are involved in test environment preparation.

Test Execution

Test execution involves running test cases developed for the system and reporting test results. The first step in test execution is generally to validate the infrastructure needed for running tests in the first place. This infrastructure primarily encompasses the test environment and test automation, including stubs that might be needed to run individual components, synthetic data used for testing or populating databases that the software needs to run, and other applications that interact with the software. The issues being sought are those that will prevent the software under test from being executed or else cause it to fail for reasons not related to faults in the software itself.

The members of the test team are responsible for test execution and reporting.

Testing Techniques

Data-Flow Analysis

Data-flow analysis can be used to increase program understanding and to develop test cases based on data flow within the program. The data-flow testing technique is based on investigating the ways values are associated with variables and the ways that these associations affect the execution of the program. Data-flow analysis focuses on occurrences of variables, following paths from the definition (or initialization) of a variable to its uses. The variable values may be used for computing values for defining other variables or used as predicate variables to decide whether a predicate is true for traversing a specific execution path. A data-flow analysis for an entire program involving all variables and traversing all usage paths requires immense computational resources; however, this technique can be applied for select variables. The simplest approach is to validate the usage of select sets of variables by executing a path that starts with definition and ends at uses of the definition. The path and the usage of the data can help in identifying suspicious code blocks and in developing test cases to validate the runtime behavior of the software. For example, for a chosen data definition-to-use path, with well-crafted test data, testing can uncover time-of-check-to-time-of-use (TOCTTOU) flaws. The ”Security Testing” section in [Howard 02] explains the data mutation technique, which deals with perturbing environment data. The same technique can be applied to internal data as well, with the help of data-flow analysis.

Code-Based Fault Injection

The fault injection technique perturbs program states by injecting software source code to force changes into the state of the program as it executes. Instrumentation is the process of non-intrusively inserting code into the software that is being analyzed and then compiling and executing the modified (or instrumented) software. Assertions are added to the code to raise a flag when a violation condition is encountered. This form of testing measures how software behaves when it is forced into anomalous circumstances. Basically this technique forces non-normative behavior of the software, and the resulting understanding can help determine whether a program has vulnerabilities that can lead to security violations. This technique can be used to force error conditions to exercise the error handling code, change execution paths, input unexpected (or abnormal) data, change return values, etc. In [Thompson 02], runtime fault injection is explained and advocated over code-based fault injection methods. One of the drawbacks of code based methods listed in the book is the lack of access to source code. However, in this content area, the assumptions are that source code is available and that the testers have the knowledge and expertise to understand the code for security implications. Refer to [Voas 98] for a detailed understanding of software fault injection concepts, methods, and tools.

Abuse Cases

Abuse cases help security testers view the software under test in the same light as attackers do. Abuse cases capture the non-normative behavior of the system. While in [McGraw 04c] abuse cases are described more as a design analysis technique than as a white box testing technique, the same technique can be used to develop innovative and effective test cases mirroring the way attackers would view the system. With access to the source code, a tester is in a better position to quickly see where the weak spots are compared to an outside attacker. The abuse case can also be applied to interactions between components within the system to capture abnormal behavior, should a component misbehave. The technique can also be used to validate design decisions and assumptions. The simplest, most practical method for creating abuse cases is usually through a process of informed brainstorming, involving security, reliability, and subject matter expertise. Known attack patterns[94] form a rich source for developing abuse cases.

Trust Boundaries Mapping

Defining zones of varying trust in an application helps identify vulnerable areas of communication and possible attack paths for security violations. Certain components of a system have trust relationships (sometimes implicit, sometime explicit) with other parts of the system. Some of these trust relationships offer ”trust elevation” possibilities—that is, these components can escalate trust privileges of a user when data or control flow cross internal boundaries from a region of less trust to a region of more trust [Hoglund 04]. For systems that have n-tier architecture or that rely on several third-party components, the potential for missing trust validation checks is high, so drawing trust boundaries becomes critical for such systems. Drawing clear boundaries of trust on component interactions and identifying data validation points (or chokepoints, as described in [Howard 02]) helps in validating those chokepoints and testing some of the design assumptions behind trust relationships. Combining trust zone mapping with data-flow analysis helps identify data that move from one trust zone to another and whether data checkpoints are sufficient to prevent trust elevation possibilities. This insight can be used to create effective test cases.

Code Coverage Analysis

Code coverage is an important type of test effectiveness measurement. Code coverage is a way of determining which code statements or paths have been exercised during testing. With respect to testing, coverage analysis helps in identifying areas of code not exercised by a set of test cases. Alternatively, coverage analysis can also help in identifying redundant test cases that do not increase coverage. During ad hoc testing (testing performed without adhering to any specific test approach or process), coverage analysis can greatly reduce the time to determine the code paths exercised and thus improve understanding of code behavior. There are various measures for coverage, such as path coverage, path testing, statement coverage, multiple condition coverage, and function coverage. When planning to use coverage analysis, establish the coverage measure and the minimum percentage of coverage required. Many tools are available for code coverage analysis. It is important to note that coverage analysis should be used to *measure* test coverage and should not be used to *create* tests. After performing coverage analysis, if certain code paths or statements were found to be not covered by the tests, the questions to ask are whether the code path should be covered and why the tests missed those paths. A risk-based approach should be employed to decide whether additional tests are required. Covering all the code paths or statements does not guarantee that the software does not have faults; however, the missed code paths or statements should definitely be inspected. One obvious risk is that unexercised code will include Trojan horse functionality, whereby seemingly

94. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack.html (Attack Patterns)

innocuous code can carry out an attack. Less obvious (but more pervasive) is the risk that unexercised code has serious bugs that can be leveraged into a successful attack [McGraw 02].

Classes of Tests

Creating security tests other than ones that directly map to security specifications is challenging, especially tests that intend to exercise the non-normative or non-functional behavior of the system. When creating such tests, it is helpful to view the software under test from multiple angles, including the data the system is handling, the environment the system will be operating in, the users of the software (including software components), the options available to configure the system, and the error handling behavior of the system. There is an obvious interaction and overlap between the different views; however, treating each one with specific focus provides a unique perspective that is very helpful in developing effective tests.

Data

All input data should be untrusted until proven otherwise, and all data must be validated as it crosses the boundary between trusted and untrusted environments [Howard 02]. Data sensitivity/criticality plays a big role in data-based testing; however, this does not imply that other data can be ignored—non-sensitive data could allow a hacker to control a system. When creating tests, it is important to test and observe the validity of data at different points in the software. Tests based on data and data flow should explore incorrectly formed data and stressing the size of the data. The section ”Attacking with Data Mutation” in [Howard 02] describes different properties of data and how to mutate data based on given properties. To understand different attack patterns relevant to program input, refer to chapter six, ”Crafting (Malicious) Input,” in [Hoglund 04]. Tests should validate data from all channels, including web inputs, databases, networks, file systems, and environment variables. Risk analysis should guide the selection of tests and the data set to be exercised.

Fuzzing

Although normally associated exclusively with black box security testing, fuzzing can also provide value in a white box testing program. Specifically, [Howard 06] introduced the concept of “smart fuzzing.” Indeed, a rigorous testing program involving smart fuzzing can be quite similar to the sorts of data testing scenarios presented above and can produce useful and meaningful results as well. [Howard 06] claims that Microsoft finds some 25-25 percent of the bugs in their code via fuzzing techniques. Although much of that is no doubt “dumb” fuzzing in black box tests, “smart” fuzzing should also be strongly considered in a white box testing program.

Environment

Software can only be considered secure if it behaves securely under all operating environments. The environment includes other systems, users, hardware, resources, networks, etc. A common cause of software field failure is miscommunication between the software and its environment [Whittaker 02]. Understanding the environment in which the software operates, and the interactions between the software and its environment, helps in uncovering vulnerable areas of the system. Understanding dependency on external resources (memory, network bandwidth, databases, etc.) helps in exploring the behavior of the software under different stress conditions. Another common source of input to programs is environment variables. If the environment variables can be manipulated, then they can have security implications. Similar conditions occur for registry information, configuration files, and property files. In general, analyzing entities outside the direct control of the system provides good insights in developing tests to ensure the robustness of the software under test, given the dependencies.

Component Interfaces

Applications usually communicate with other software systems. Within an application, components interface with each other to provide services and exchange data. Common causes of failure at interfaces are misunderstanding of data usage, data lengths, data validation, assumptions, trust relationships, etc. Understanding the interfaces exposed by components is essential in exposing security bugs hidden in the interactions between components. The need for such understanding and testing becomes paramount when third-party software is used or when the source code is not available for a particular component. Another important benefit of understanding component interfaces is validation of principles of compartmentalization. The basic idea behind compartmentalization is to minimize the amount of damage that can be done to a system by breaking up the system into as few units as possible while still isolating code that has security privileges [McGraw 02]. Test cases can be developed to validate compartmentalization and to explore failure behavior of components in the event of security violations and how the failure affects other components.

Configuration

In many cases, software comes with various parameters set by default, possibly with no regard for security. Often, functional testing is performed only with the default settings, thus leaving sections of code related to non-default settings untested. Two main concerns with configuration parameters with respect to security are storing sensitive data in configuration files and configuration parameters changing the flow of execution paths. For example, user privileges, user roles, or user passwords are stored in the configuration files, which could be manipulated to elevate privilege, change roles, or access the system as a valid user. Configuration settings that change the path of execution could exercise vulnerable code sections that were not developed with security in mind. The change of flow also applies to cases where the settings are changed from one security level to another, where the code sections are developed with security in mind. For example, changing an endpoint from requiring authentication to *not* requiring authentication means the endpoint can be accessed by everyone. When a system has multiple configurable options, testing all combinations of configuration can be time consuming; however, with access to source code, a risk-based approach can help in selecting combinations that have higher probability in exposing security violations. In addition, coverage analysis should aid in determining gaps in test coverage of code paths.

Error handling

The most neglected code paths during the testing process are error handling routines. Error handling in this paper includes exception handling, error recovery, and fault tolerance routines. Functionality tests are normally geared towards validating requirements, which generally do not describe negative (or error) scenarios. Even when negative functional tests are created, they don’t test for non-normative behavior or extreme error conditions, which can have security implications. For example, functional stress testing is not performed with an objective to break the system to expose security vulnerability. Validating the error handling behavior of the system is critical during security testing, especially subjecting the system to unusual and unexpected error conditions. Unusual errors are those that have a low probability of occurrence during normal usage. Unexpected errors are those that are not explicitly specified in the design specification, and the developers did not think of handling the error. For example, a system call may throw an ”unable to load library” error, which may not be explicitly listed in the design documentation as an error to be handled. All aspects of error handling should be verified and validated, including error propagation, error observability, and error recovery. Error propagation is how the errors are propagated through the call chain. Error observability is how the error is identified and what parameters are passed as error messages. Error recovery is getting back to a state conforming to specifications. For example, return codes for errors may not be checked, leading to uninitialized variables and garbage data in buffers; if the memory is manipulated before causing a failure, the uninitialized memory may contain attacker-supplied data. Another common mistake to look for is when sensitive information is included as part of the error messages.

Tools

Source Code Analysis

Source code analysis is the process of checking source code for coding problems based on a fixed set of patterns or rules that might indicate possible security vulnerabilities. Static analysis tools scan the source code and automatically detect errors that typically pass through compilers and become latent problems. The strength of static analysis depends on the fixed set of patterns or rules used by the tool; static analysis does not find all security issues. The output of the static analysis tool still requires human evaluation. For white box testing, the results of the source code analysis provide a useful insight into the security posture of the application and aid in test case development. White box testing should verify that the potential vulnerabilities uncovered by the static tool do not lead to security violations. Some static analysis tools provide data-flow and control-flow analysis support, which are useful during test case development.

A detailed discussion about the analysis or the tools is outside the scope of this content area. This section addresses how source code analysis tools aid in white box testing. For a more detailed discussion on the analysis and the tools, refer to the Source Code Analysis[116] content area.

Program Understanding Tools

In general, white box testers should have access to the same tools, documentation, and environment as the developers and functional testers on the project do. In addition, tools that aid in program understanding, such as software visualization, code navigation, debugging, and disassembly tools, greatly enhance productivity during testing.

Coverage Analysis

Code coverage tools measure how thoroughly tests exercise programs. There are many different coverage measures, including statement coverage, branch coverage, and multiple-condition coverage. The coverage tool would modify the code to record the statements executed and which expressions evaluate which way (the true case or the false case of the expression). Modification is done either to the source code or to the executable the compiler generates. There are several commercial and freeware coverage tools available. Coverage tool selection should be based on the type of coverage measure selected, the language of the source code, the size of the program, and the ease of integration into the build process.

Profiling

Profiling allows testers to learn where the software under test is spending most of its time and the sequence of function calls as well. This information can show which pieces of the software are slower than expected and which functions are called more often than expected. From a security testing perspective, the knowledge of performance bottlenecks help uncover vulnerable areas that are not apparent during static analysis. The call graph produced by the profiling tool is helpful in program understanding. Certain profiling tools also detect memory leaks and memory access errors (potential sources of security violations). In general, the functional testing team or the development team should have access to the profiling tool, and the security testers should use the same tool to understand the dynamic behavior of the software under test.

Results to Expect

Any security testing method aims to ensure that the software under test meets the security goals of the system and is robust and resistant to malicious attacks. Security testing involves taking two diverse approaches: one, testing security mechanisms to ensure that their functionality is properly implemented; and two, performing risk-based security testing motivated by understanding and simulating the attacker's approach. White box security testing follows both these approaches and uncovers programming and implementation errors. The types of errors uncovered during white box testing are several and are very context sensitive to the software under test. Some examples of errors uncovered include

- data inputs compromising security
- sensitive data being exposed to unauthorized users
- improper control flows compromising security
- incorrect implementations of security functionality
- unintended software behavior that has security implications
- design flaws not apparent from the design specification
- boundary limitations not apparent at the interface level

White box testing greatly enhances overall test effectiveness and test coverage. It can greatly improve productivity in uncovering bugs that are hard to find with black box testing or other testing methods alone.

116. http://buildsecurityin.us-cert.gov/bsi/articles/tools/code.html (Source Code Analysis)

Business Case

The goal of security testing is to ensure the robustness of the software under test, even in the presence of a malicious attack. The designers and the specification might outline a secure design, the developers might be diligent and write secure code, but it’s the testing process that determines whether the software is secure in the real world. Testing is an essential form of assurance. Testing is laborious, time consuming, and expensive, so the choice of testing (black box, or white box, or a combination) should be based on the risks to the system. Risk analysis provides the right context and information to make tradeoffs between time and effort to achieve test effectiveness.

White box testing is typically very effective in validating design decisions and assumptions and finding programming errors and implementation errors in software. For example, an application may use cryptography to secure data from specific threats, but an implementation error such as a poor key management technique can still leave the application vulnerable to security violations. White box testing can uncover such implementation errors.

Benefits

There are many benefits to white box testing, including the following:

- Analyzing source code and developing tests based on the implementation details enables testers to find programming errors quickly. For example, a white box tester looking at the implementation can quickly uncover a way, say, through an error handling mechanism, to expose secret data processed by a component. Finding such vulnerabilities through black box testing require comparatively more effort than found through white box testing. This increases the productivity of testing effort.
- Executing some (hard to set up) black box tests as white box tests reduces complexity in test setup and execution. For example, to drive a specific input into a component, buried inside the software, may require elaborate setup for black box testing but may be done more directly through white box testing by isolating the component and testing it on its own. This reduces the overall cost (in terms of time and effort) required to perform such tests.
- Validating design decisions and assumptions quickly through white box testing increases effectiveness. The design specification may outline a secure design, but the implementation may not exactly capture the design intent. For example, a design might outline the use of protected channels for data transfer between two components, but the implementation may be using an unprotected method for temporary storage before the data transfer. This increases the productivity of testing effort.
- Finding ”unintended” features can be quicker during white box testing. Security testing is not just about finding vulnerabilities in the intended functionality of the software but also about examining unintended functionality introduced during implementation. Having access to the source code improves understanding and uncovering the additional unintended behavior of the software. For example, a component may have additional functions exposed to support interactions with some other component, but the same functions may be used to expose protected data from a third component. Depending on the nature of the ”unintended” functionality, it may require a lot more effort to uncover such problems through black box testing alone.

Costs

The main cost drivers for white box testing are the following:

- specialized skill requirements: White box testing is knowledge intensive. White box testers should not only know how to analyze code for security issues but also understand different tools and techniques to test the software. Security testing is not just validating designed functionality but also proving that the defensive mechanisms work correctly. This requires invaluable experience and expertise. Testers who can perform such tasks are expensive and hard to get.
- support software development and tools: White box testing requires development of support software and tools to perform testing. Both the support software and the tools are largely based on the context

of the software under test and the type of test technique employed. The type of tools used includes program understanding tools, coverage tools, fault injection tools, and source code analyzers.

- analysis and testing time: White box testing is time consuming, especially when applied to the whole system. Analyzing design and source code in detail for security testing is time consuming, but is an essential part of white box testing. Tools (source code analyzers, debuggers, etc.) and program understanding techniques (flow graphs, data-flow graphs, etc.) help in speeding up analysis. White box testing directly identifies implementation bugs, but whether the bugs can be exploited requires further analysis work. The consequences of failure help determine the amount of testing time and effort dedicated to certain areas.

Alternatives

There are alternatives to white box testing:

- White box testing can complement black box testing to increase overall test effectiveness. Based on risk assessment, certain areas of the software may require more scrutiny than others. White box testing could be performed for specific high-risk areas, and black box testing could be performed for the whole system. By complementing the two testing methods, more tests can be developed, focusing on both implementation issues and usage issues.
- Gray box testing can be used to combine both white box and black box testing methods in a powerful way. In a typical case, white box analysis is used to find vulnerable areas, and black box testing is then used to develop working attacks against these areas. The white box analysis increases productivity in finding vulnerable areas, while the black box testing method of driving data inputs decreases the cost of test setup and test execution.

All testing methods can reveal possible software risks and potential exploits. White box testing directly identifies more bugs in the software. White box testing is time consuming and expensive and requires specialized skills. As with any testing method, white box testing has benefits, associated costs, and alternatives. An effective testing approach balances efficiency and effectiveness in order to identify the greatest number of critical defects for the least cost.

Skills and Training

White box testing requires knowledge of software security design and coding practices, an understanding of an attacker’s mindset, knowledge of known attack patterns, vulnerabilities and threats, and the use of different testing tools and techniques. White box testing brings together the skills of a security developer, an attacker, and a tester.

Books like Writing Secure Code [Howard 02], 19 Deadly Sins [Howard 05], and Building Secure Software [Viega 02] help educate software professionals on how to write secure software, and books like How to Break Software Security [Whittaker 03] and Exploiting Software [Hoglund 04] help educate professionals on how to think like an attacker. There are several good books on software testing, including the two classics Software Testing Techniques [Beizer 90] and Testing Computer Software [Kaner 99]. There are several valuable information sites that detail known vulnerabilities, attack patterns, security tools, etc. White box testing is knowledge intensive and relies on expertise and experience.

Case Study

A large merchant organization involved in online business was in the process of developing an online e-commerce web site. In an effort to allow customers to electronically and efficiently transfer funds from customer checking accounts to merchant accounts, the merchant organization had outsourced its payment processing to a third-party Internet-enabled financial transaction payment firm. The third-party payments software provided customized interfaces to facilitate payment processing between the customers and the merchant organization.

A high-level security risk analysis was conducted on the system. Risk assessment identified transactions processing between the payments interface and the application as one of the risks. The impact of fraudulent transactions is serious for both the customers and for the merchant organization. The customers could suffer significant financial loss and hardships resulting from unauthorized transactions that could deplete account balances. The credibility and the reputation of the merchant organization could be severely damaged as a result of fraudulent transactions should they become publicized.

A thorough white box testing was conducted on the modules using the payments interface. First, all the component interfaces were identified and illustrated as interface diagrams; second, trust relationship boundaries were drawn on the component interactions; and third, data flows between the components were drawn. Abuse cases were developed based on this information. One of the abuse cases pointed to exercising the payments processing functionality as an anonymous user. The trust relationship mapping and data flow showed a path where inputs from the users were not validated or authenticated. A test case was developed to submit an account transfer from an external account to the merchant account anonymously and the account transfer was completed successfully, which is a critical software failure: unauthorized transactions via unauthenticated channel were allowed.

Risk assessment also identified a weak authentication component in the payment customer service component of the system. As in the case above, trust relationship boundaries and data-flow analysis were conducted on this component. After analysis and testing, it was shown that an attacker could gain direct access to the merchant administrative accounts. With this access, an attacker could redirect transactions from a merchant account to another, non-merchant account.

Using the two exploits described above, white box testers showed that an attacker could funnel payments from an unwitting customer to the merchant account and then redirect the transaction from the merchant account to a non-merchant account. The two bugs put together form a serious breach of security with significant business impact.

This example shows that risk analysis yields business-relevant results and points to specific vulnerable areas of the software. Subsequent to risk analysis, performing white box testing in concentrated areas aids in quickly uncovering design assumptions and implementation errors. This example also shows that finding an exploit does not mean that the job is done. Collecting information about various bugs in the software and analyzing them together shows how multiple exploits can be stringed together to form a full attack.

Conclusion

White box testing for security is useful and effective. It should follow a risk-based approach to balance the testing effort with consequences of software failure. Architectural and design-level risk analysis provide the right context to plan and perform white box testing. White box testing can be used with black box testing to improve overall test effectiveness. It uncovers programming and implementation errors.

This paper introduces a risk-based approach and tools and techniques applicable to white box testing for security. Concepts and knowledge from two areas, traditional white box testing and security-based testing, were brought together. Other content areas on this web portal discuss different aspects of software security in detail. To gain more in-depth understanding of white box testing, readers are urged to supplement the knowledge gained here with other related areas available on this web site. The links to related topics are given in the Links section.

Links

Best Practices: Security Testing[166]

Knowledge: Attack Patterns[167]

166. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing.html (Security Testing)
167. http://buildsecurityin.us-cert.gov/bsi/articles/knowledge/attack.html (Attack Patterns)

Tools: Black Box Testing[168]

Tools: Code Analysis[169]

Best Practices: Architectural Risk Analysis[170]

Glossary

ad hoc testing	Testing[172] carried out using no recognized test case design technique[173]. [BS-7925]
authentication	Authentication is the process of confirming the correctness of the claimed identity. [SANS 03]
black box testing	Testing that is based on an analysis of the specification[174] of the component[175] without reference to its internal workings. [BS-7925]
buffer overflow	A buffer overflow occurs when a program or process tries to store more data in a data storage area than it was intended to hold. Since buffers are created to contain a finite amount of data, the extra information —which has to go somewhere—can overflow into the runtime stack, which contains control information such as function return addresses and error handlers.
bug	See fault[176].
component	A minimal software item for which a separate specification[177] is available. [BS-7925]
correctness	The degree to which software conforms to its specification[178]. [BS-7925]
cryptography	Cryptography garbles a message in such a way that anyone who intercepts the message cannot understand it. [SANS 03]
dynamic analysis	The process of evaluating a system or component[179] based on its behavior[180] during execution. [IEEE 90]
encryption	Cryptographic transformation of data (called ”plaintext”) into a form (called ”cipher text”) that conceals the data’s original meaning to prevent it from being known or used. [SANS 03]
failure	The inability of a system or component to perform its required functions within specified performance requirements. [IEEE 90]
fault	A manifestation of an error[181] in software. A fault[182], if encountered, may cause a failure[183]. [DO-178B]

168. http://buildsecurityin.us-cert.gov/bsi/articles/tools/black-box.html (Black Box Testing)
169. http://buildsecurityin.us-cert.gov/bsi/articles/tools/code.html (Source Code Analysis)
170. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/architecture.html (Architectural Risk Analysis)

integration testing	Testing[184] performed to expose fault[185]s in the interfaces and in the interaction between integrated component[186]s. [BS-7925]
interface testing	Integration testing [187]in which the interfaces between system component[188]s are tested. [BS-7925]
National Institute of Standards and Technology (NIST)	A unit of the U.S. Commerce Department. Formerly known as the National Bureau of Standards, NIST promotes and maintains measurement standards. It also has active programs for encouraging and helping industry and science to develop and use these standards. [SANS 03]
negative requirements	Requirements that state what software should not do.
operational testing	Testing [189]conducted to evaluate a system or component[190] in its operational environment. [IEEE 90]
path testing	Basis path structured testing uses the control flow structure of software to establish path coverage criteria. The resultant test sets provide more thorough testing than statement and branch coverage [Watson 96].
precondition	Environmental and state conditions that must be fulfilled before the component can be executed with a particular input value.
race condition	A race condition exploits the small window of time between a security control being applied and the service being used. [SANS 03]
regression testing	Retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made. [BS-7925]
requirement	A capability that must be met or possessed by the system/software (requirements may be functional or non-functional). [BS-7925]
requirements-based testing	Designing tests based on objectives derived from requirements for the software component (e.g., tests that exercise specific functions or probe the non-functional constraints such as performance or security). [BS-7925]
reverse engineering	Acquiring sensitive data by disassembling and analyzing the design of a system component [SANS 03]; acquiring knowledge of a binary program’s algorithms or data structures.
risk assessment	The process by which risks are identified and the impact of those risks is determined. [SANS 03]
security policy	A set of rules and practices that specify or regulate how a system or organization provides security

	services to protect sensitive and critical system resources. [SANS 03]
specification	A description, in any suitable form, of requirements. [BS-7925]
specification testing	An approach to testing wherein the testing is restricted to verifying that the system/software meets the specification. [BS-7925]
SQL injection	SQL injection is a type of input validation attack specific to database-driven applications where SQL code is inserted into application queries to manipulate the database. [SANS 03]
static analysis	Analysis of a program carried out without executing the program. [BS-7925]
stress testing	Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements. [IEEE 90]
stub	A skeletal or special-purpose implementation of a software module used to develop or test a component that calls or is otherwise dependent on it. [IEEE 90].
system testing	The process of testing an integrated system to verify that it meets specified requirements. [Hetzel 88]
test automation	The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.
test case	A set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. [IEEE 90]
test suite	A collection of one or more test cases for the software under test. [BS-7925]
test driver	A program or test tool used to execute software against a test suite. [BS-7925]
test environment	A description of the hardware and software environment in which tests will be run and any other software with which the software under test interacts when under test, including stubs and test drivers. [BS-7925]
test plan	A record of the test planning process detailing the degree of tester independence, the test environment, the test case design techniques and test measurement techniques to be used, and the rationale for their choice. [BS-7925]
vulnerability	A defect or weakness in a system's design, implementation, or operation and management that

could be exploited to violate the system's security policy. [SANS 03]

Bibliography

[Beizer 90]	Beizer, Boris. *Software Testing Techniques*. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).
[Binder 99]	Binder, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Addison-Wesley Object Technology Series). Boston, MA: Addison-Wesley Professional, 1999.
[Chess 04]	Chess, Brian & McGraw, Gary. "Static Analysis for Security." IEEE Security & Privacy 2, 6 (Nov.-Dec. 2004): 76-79.
[Fewster 99]	Fewster, Mark & Graham, Doroty. *Software Test Automation*. Boston, MA: Addison-Wesley Professional, 1999.
[Graff 03]	Graff, Mark G. & Van Wyk, Kenneth R. *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly, 2003.
[Hetzel 88]	Hetzel, William C. *The Complete Guide to Software Testing, 2nd ed*. Wellesley, MA: QED Information Sciences, 1988.
[Hoglund 04]	Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley Professional, 2004.
[Howard 02]	Howard, Michael & LeBlanc, David. *Writing Secure Code, 2nd ed*. Redmond, WA: Microsoft Press, 2002.
[Howard 05]	Howard, Michael; LeBlanc, David; & Viega, John. *19 Deadly Sins of Software Security*. Emeryville, CA: McGraw-Hill/Osborne Media, 2005.
[Howard 06]	Howard, Michael & Lipner, Steve. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006, ISBN 0735622142.
[IEEE 90]	IEEE. *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990). Los Alamitos, CA: IEEE Computer Society Press, 1990.
[Kaner 99]	Kaner, Cem; Falk, Jack; & Nguyen, Hung Quoc. *Testing Computer Software, 2nd ed*. New York, NY: John Wiley & Sons, 1999.
[Marciniak 94]	Marciniak, John J., ed. *Encyclopedia of Software Engineering*, 131-165. New York, NY: John Wiley & Sons, 1994.
[Marick 94]	Marick, Brian. *The Craft of Software Testing: Subsystems Testing Including Object-Based and*

	Object-Oriented Testing. Upper Saddle River, NJ: Prentice Hall PTR, 1994.
[Marick 97]	Marick, Brian. *How to Misuse Code Coverage*[193]. (1997).
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software*. Boston, MA: Addison Wesley, 2002.
[McGraw 04a]	McGraw, Gary & Potter, Bruce. "Software Security Testing." IEEE Security and Privacy 2, 5 (Sept.-Oct. 2004): 81-85.
[McGraw 04b]	McGraw, Gary. "Application Security Testing Tools: Worth the Money?[194]" *Network Magazine*, November 1, 2004.
[McGraw 04c]	McGraw, Gary; Anton, Annie I.; & Hope, Paco. "Misuse and Abuse Cases: Getting Past the Positive." *IEEE Security & Privacy 2*, 3 (March-April 2004): 32-34.
[Redmill 04]	Redmill, Felix. "Exploring Risk-Based Testing and Its Implications." *Software Testing, Verification and Reliability 14*, 1 (March 2004): 3-15.
[Thompson 02]	Thompson, Herbert H.; Whittaker, James A.; & Mottay, Florence E. "Software Security Vulnerability Testing in Hostile Environments," 260-264. *Proceedings of the 2002 ACM Symposium on Applied Computing*. Madrid, Spain, 2002. New York, NY: ACM Press, 2002.
[Tonella 04]	Tonella, Paolo & Ricca, Filippo. "A 2-Layer Model for the White-Box Testing of Web Applications," 11-19. *Proceedings of the Sixth IEEE International Workshop on Web Site Evolution (WSE'04)*. September 11, 2004. Los Alamitos, CA: IEEE Computer Society Press, 2004.
[Viega 03]	Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++*. Sebastopol, CA: O'Reilly, 2003 (ISBN 0596003943).
[Verdon 04]	Verdon, Denis & McGraw, Gary. "Risk Analysis in Software Design." *IEEE Security & Privacy 2*, 4 (July-Aug. 2004): 79-84.
[Voas 98]	Voas, Jeffrey M. & McGraw, Gary. *Software Fault Injection: Inoculating Programs Against Errors*, 47-48. New York, NY: John Wiley & Sons, 1998.
[Watson 96]	Watson, Arthur H. & McCabe, Thomas J. *Structured Testing*[195]*: A Testing Methodology Using the Cyclomatic Complexity Metric* (NIST Special Publication 500-235). Gaithersburg, MD: National Institute of Standards and Technology, 1996.
[Whittaker 02]	Whittaker, J. A. *How to Break Software*. Reading, MA: Addison Wesley, 2002.

[Whittaker 03]	Whittaker, J. A. & Thompson, H. H. *How to Break Software Security*. Reading MA: Addison Wesley, 2003.
[Wysopal 03]	Wysopal, Chris; Nelson, Lucas; Zovi, Dino Dai; & Dustin, Elfriede. *The Art of Software Security Testing*. Upper Saddle River, NJ: Pearson Education, Inc.

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com